852L1-DataCoding-Lecture9

December 4, 2020

1 Lecture 9: Data Science with Python

In our previous lectures, we have learnt the basics of programming with Python: data types, control flow (loops and conditional statements), and functions.

In this lecture, we want to put our Python knowledge to practical use and strengthen one of the most fundamental skills in Economics: **Data Analysis**.

2 Welcome Pandas, the library for data analysis

In the data science community, **Pandas** has emerged as the reference library for data analysis within the Python ecosystem.

According to the official website:

> **Pandas** is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Pandas allows us to import a dataset from a standard Comma-Separated Values (.csv) file or Excel file, and convert it into a Python object called **data frame** packed with attributes and methods specifically designed for data analysis.

Pandas is the perfect tool for data reading, data manipulation, aggregation, and visualisation. Among many other features, Pandas allows economists to

- Access and read data via indexing
- Find data that meets certain conditions via filtering
- Update, add, or delete data
- Compute main descriptive statistics
- Aggregate and group data to find patterns

Importing and reading data

The very first thing that we want to do is to import the library itself. If you are using Python via the Anaconda distribution, Pandas should already be installed and ready to use.

[1]:
```
# Import pandas library using the conventional alias
import pandas as pd
```

Great, Pandas is now loaded!

Next we want to load an example dataset to understand how Pandas handles it. As we will see, Pandas creates a **data frame** object from raw data.

Pandas can import datasets from many formats, including

- Comma-separated values (csv) files
- Excel files (xls, xlsx)
- Stata data files (dta)

and many more.

Each format has its own Pandas method, i.e. function, to be used in order to import the dataset and convert it into a data frame. A complete list of importing methods can be found here.

In this lecture, we will work a **comma-separated values** file containing data on expenditure and other variables collected from a survey of British households.

To import the CSV file, we need to use the `pd.read_csv()` method which syntax is

`df_name = pd.read_csv("path/file_name.csv")`

where `path` indicates the location and `file_name` the name of the CSV file containing your data.

[2]:
```
# Import household expenditure dataset from CSV files and create a data frame␣
↪object
df_exp = pd.read_csv("expenditure_data.csv")
```

Pandas has now read the CSV file and created a new **data frame** object named `df_exp`. To confirm this, let's quickly check the data type of the newly created variable:

[3]:
```
# Check data type of imported dataset
type(df_exp)
```

[3]:
```
pandas.core.frame.DataFrame
```

Simply put, a **data frame** can be seen as a table with rows and columns, where

- Each **column** represents a different **variable**
- Each **row** represents a different **observation**

To see this first-hand, let's print out our expenditure survey data frame.

[4]:
```
# Display the expenditure survey data frame
df_exp
```

[4]:
```
      expenditure   income   maininc    region   nadults     nkids  SexHRP  \
0        380.6958  465.360  earnings  East Mid  2 adults  Two or m  Female
1        546.4134  855.260  earnings    London  2 adults  No child  Female
2        242.1890  160.960  earnings  South Ea   1 adult  No child  Female
3        421.3824  656.220  earnings   Eastern  2 adults  No child    Male
4        370.4056  398.800  earnings  South Ea   1 adult  No child    Male
…             …        …        …         …        …         …        …
5139     482.4708  782.040  earnings  North We  2 adults  No child  Female
```

```
5140     282.6099   612.272  other so  West Mid  2 adults  No child    Male
5141     934.1562  1134.920  earnings     Wales   1 adult  Two or m  Female
5142     426.5105   663.669  other so  North We  2 adults  No child    Male
5143     700.0040  1076.870  earnings  Scotland  3 adults  No child    Male

       housing  internet
0     Public r         1
1        Owned         1
2        Owned         1
3        Owned         1
4        Owned         1
…          …         …
5139   Private         1
5140     Owned         1
5141     Owned         1
5142     Owned         1
5143     Owned         1

[5144 rows x 9 columns]
```

The advantage of data frames over simple tables is that they come with lots of functionalities that allow us to read, access, and modify the data contained in a data frame.

For instance, we can visually inspect the **first** and **last** observations in the data frame using the `head()` and `tail()` methods.

[5]:
```python
# Display first 10 observations
df_exp.head(10)
```

[5]:
```
   expenditure    income   maininc    region   nadults     nkids  SexHRP  \
0     380.6958   465.360  earnings  East Mid  2 adults  Two or m  Female
1     546.4134   855.260  earnings    London  2 adults  No child  Female
2     242.1890   160.960  earnings  South Ea   1 adult  No child  Female
3     421.3824   656.220  earnings   Eastern  2 adults  No child    Male
4     370.4056   398.800  earnings  South Ea   1 adult  No child    Male
5     172.3972   321.020  earnings  Scotland   1 adult  No child    Male
6     202.4250   350.834  other so  Northern  2 adults  No child    Male
7     730.6580  1184.990  earnings  South Ea  4 and mo  No child  Female
8     361.9310   619.570  earnings     Wales  2 adults  No child    Male
9     659.9117  1117.842  other so  Yorkshir  2 adults  No child    Male

    housing  internet
0  Public r         1
1     Owned         1
2     Owned         1
3     Owned         1
4     Owned         1
5  Public r         0
```

```
6     Owned          1
7  Public r          1
8     Owned          1
9     Owned          1
```

[6]:
```python
# Display last 15 observations
df_exp.tail(15)
```

[6]:
```
       expenditure     income   maininc    region   nadults     nkids  SexHRP  \
5129    1175.00000  1184.9900  earnings  South Ea  3 adults  One chil    Male
5130      83.81268    84.6625  other so  West Mid   1 adult  No child    Male
5131     560.71360  1184.9900  earnings     Wales  3 adults  One chil    Male
5132     140.16690   297.6000  other so    London   1 adult  Two or m  Female
5133     241.42590   201.9300  earnings  North We   1 adult  No child  Female
5134     449.14870   137.1420  other so  North We   1 adult  No child  Female
5135     509.64130  1184.9900  earnings     Wales  4 and mo  No child    Male
5136     164.50280   246.7820  other so  North We   1 adult  No child  Female
5137     244.63870   858.2250  earnings     Wales   1 adult  No child    Male
5138     155.61500   215.3800  other so  South Ea   1 adult  Two or m  Female
5139     482.47080   782.0400  earnings  North We  2 adults  No child  Female
5140     282.60990   612.2720  other so  West Mid  2 adults  No child    Male
5141     934.15620  1134.9200  earnings     Wales   1 adult  Two or m  Female
5142     426.51050   663.6690  other so  North We  2 adults  No child    Male
5143     700.00400  1076.8700  earnings  Scotland  3 adults  No child    Male

       housing  internet
5129     Owned         1
5130     Owned         0
5131     Owned         1
5132  Public r         1
5133   Private         1
5134     Owned         1
5135     Owned         1
5136     Owned         0
5137     Owned         0
5138   Private         1
5139   Private         1
5140     Owned         1
5141     Owned         1
5142     Owned         1
5143     Owned         1
```

Or we can check the number of variables and observations with the `shape` attribute

[7]:
```python
# Get number of observations and variables
nobs, nvars = df_exp.shape
print(f"This dataset contains {nvars} variables and {nobs} observations")
```

```
This dataset contains 9 variables and 5144 observations
```

As well as the names of the variables in our data frame with the `columns` index.

```
[8]: # Get variable names in a list
var_names = df_exp.columns.to_list()

# Display them nicely
for pos, var_name in enumerate(var_names):
    print(pos+1, var_name)
```

```
1 expenditure
2 income
3 maininc
4 region
5 nadults
6 nkids
7 SexHRP
8 housing
9 internet
```

Working with data frames

Data frames are the main data type of Pandas.

The easiest way to understand the basic features of data frames is to think of them as **dictionaries containing lists**. Data frames are **much more** than simple dictionaries, however this analogy will allow us to grasps a few key concepts.

As always, let's work with a minimal example to keep things simple. For instance, consider a dictionary that contains information on a few tennis players.

```
[9]: # Create a dictionary of lists representing a dataset
players = {
    "name": ["Luca", "Roger", "Rafael", "Novak"],
    "surname": ["Rondina", "Federer", "Nadal", "Djokovic"],
    "email": ["Luca.Rondina@surrey.ac.uk", "R.Federer@tennis.com", "R.
 ↪Nadal@tennis.com", "N.Djokovic@tennis.com"]
}
```

The dictionary above was carefully created in a way that keys and values have a natural interpretation:

- The keys represent different variables in the dataset
- The values contain a list of observations

As a result, accessing and updating data is done via the familiar indexing of the dictionary.

```
[10]: # Display all email addresses
players["email"]

# Update Luca's email address
```

```
players["email"][0] = "Luca.Rondina@sussex.ac.uk"
```

Data frames work in a very similar way. Let's create a data frame from the above dictionary and see how we can access and update information within a data frame.

[11]:
```
# Create a data frame from a given dictionary
df_players = pd.DataFrame(players)

# Display the data frame
df_players
```

[11]:

	name	surname	email
0	Luca	Rondina	Luca.Rondina@sussex.ac.uk
1	Roger	Federer	R.Federer@tennis.com
2	Rafael	Nadal	R.Nadal@tennis.com
3	Novak	Djokovic	N.Djokovic@tennis.com

[12]:
```
# Display all email addresses
df_players["email"]
```

[12]:
```
0    Luca.Rondina@sussex.ac.uk
1         R.Federer@tennis.com
2           R.Nadal@tennis.com
3        N.Djokovic@tennis.com
Name: email, dtype: object
```

[13]:
```
# Update Luca's email address
df_players["email"][0] = "L.Rondina@sussex.ac.uk"
df_players
```

[13]:

	name	surname	email
0	Luca	Rondina	L.Rondina@sussex.ac.uk
1	Roger	Federer	R.Federer@tennis.com
2	Rafael	Nadal	R.Nadal@tennis.com
3	Novak	Djokovic	N.Djokovic@tennis.com

2.1 Accessing data

The first advantage of data frames over dictionaries is that we can access multiple data at the same time and get a selected **view** of the data frame.

Let's say that we want to see the name and surname of the players, but not their email address. We can do that by indexing the data frame with a **list of variable names**.

[14]:
```
# Display name and surname of the players only
df_players[["name", "surname"]]
```

[14]:
```
      name   surname
0     Luca   Rondina
1    Roger   Federer
2  Rafael      Nadal
3    Novak  Djokovic
```

The interesting thing is that the operation above is **returning a data frame**! Therefore, any data frame method can be directly applied to the selected view of the data frame directly.

Of course, this works on our expenditure survey data frame `df_exp` as well. If we want to see the first five observations of expenditure and income only, we can do so with the following lines of code.

[15]:
```python
# Display the first five observations of expenditure and income only
selected_vars = ["expenditure", "income"]
df_exp[selected_vars].head()                       # With no arguments head()␣
 ↪defaults to head(5)
```

[15]:
```
   expenditure  income
0     380.6958  465.36
1     546.4134  855.26
2     242.1890  160.96
3     421.3824  656.22
4     370.4056  398.80
```

Or we can get information on how many British households have access to internet

[16]:
```python
# Apply the value_counts() method to internet variable
df_exp["internet"].value_counts()

# 1 is access to internet, 0 is no access to internet
```

[16]:
```
1    4232
0     912
Name: internet, dtype: int64
```

Accessing specific observations

Another important feature of data frames is the ability to access, and potentially modify, specific observations (or subsets) in the dataset.

The easiest (and best) way to do this is by using the `loc()` method which access elements in the data frame via **two** labels:

- A first label (or list of labels) to select the row(s)
- A second label (or list of labels) to select the column(s)

The general syntax for loc is

`df_name.loc[rows_label, columns_label]`

where `rows_label` and `columns_label` can either be a single label or a list of labels.

Let's use our players dataframe for demonstration purposes:

```
[17]: df_players
```

[17]:

	name	surname	email
0	Luca	Rondina	L.Rondina@sussex.ac.uk
1	Roger	Federer	R.Federer@tennis.com
2	Rafael	Nadal	R.Nadal@tennis.com
3	Novak	Djokovic	N.Djokovic@tennis.com

If we want to access the name and surname of Federer and Nadal only, we can do that by using the row labels and the variable names as follows

```
[18]: df_players.loc[[1,2], ["name", "surname"]]
```

[18]:

	name	surname
1	Roger	Federer
2	Rafael	Nadal

Notice that **slicing** also works. Differently from list indexing though, **both** start and end indexes are **included**.

```
[19]: df_players.loc[1:2, "name":"surname"]
```

[19]:

	name	surname
1	Roger	Federer
2	Rafael	Nadal

Moreover, the resulting view of the data frame can be stored in a variable for later use if needed.

```
[20]: df_roger_rafa = df_players.loc[1:2, "name":"surname"]
```

Filtering data frames

One of the most common tasks when exploring datasets is to search for a subset of observations that meet certain conditions and perform some operations on them. This is known as **data filtering**.

A few examples:

- How many British households have a weekly expenditure between £200 and £300?
- What is the median weekly income in Wales?
- What is the percentage of households in London with no children?

Data filtering can be done by via a three step approach:

1. Create a **series of Boolean (True or False) values** that indicates which subset of observations meet certain conditions
2. Extract the subset of observation via **Boolean indexing**
3. Perform some operation

Let's start simple and check whether there is a player whose first name is Luca in our players data frame:

```
[21]: # Create a Boolean series that checks whether the players name are equal to
      ↪"Luca"
      name_is_luca = (df_players["name"] == "Luca")
      name_is_luca
```

```
[21]: 0     True
      1    False
      2    False
      3    False
      Name: name, dtype: bool
```

As you can see, Pandas has created a series of True or False values that indicates whether each observation matches the condition specified. In our case, Pandas went through all names in the data frame and checked whether they were equal to Luca.

We can now use this series and filter out the observations that match the condition using Boolean indexing:

```
[22]: df_players.loc[name_is_luca]
```

[22]:

	name	surname	email
0	Luca	Rondina	L.Rondina@sussex.ac.uk

Similarly to conditional expressions in Python, we can chain multiple conditions using Boolean operators.

However, Pandas uses different symbols for the `and`, `or`, and `not` operators:

Operator	Usage	Result
`&`	condition1 `&` condition2	`True` if both conditions are `True`, otherwise `False`
`|`	condition1 `|` condition2	`True` if at least one of the conditions is `True`, otherwise `False`
`~`	`~` condition	`True` if condition is `False`, and vice versa

For instance:

[23]:
```python
# Check whether player name is Luca or Roger
name_is_luca_or_roger = (df_players["name"] == "Luca") | (df_players["name"] ==
↪"Roger")

# Filter data frame
df_players.loc[name_is_luca_or_roger]
```

[23]:

	name	surname	email
0	Luca	Rondina	L.Rondina@sussex.ac.uk
1	Roger	Federer	R.Federer@tennis.com

[24]:
```python
# Check whether player name is Luca and surname Federer
fullname_is_lucafederer = (df_players["name"] == "Luca") &
↪(df_players["surname"] == "Federer")
fullname_is_lucafederer

# Filter data frame
df_players.loc[fullname_is_lucafederer]
```

[24]:
```
Empty DataFrame
Columns: [name, surname, email]
Index: []
```

We can also filter data that does NOT meet a condition and inspect specific variables.

[25]:
```python
# Get emails of players whose name is not Luca or Roger
df_players.loc[~name_is_luca_or_roger, "email"]
```

[25]:
```
2       R.Nadal@tennis.com
3    N.Djokovic@tennis.com
Name: email, dtype: object
```

3 Updating data frames

Another common task in data analysis is updating the existing dataset to either reflect new information or to incorporate information from different sources (merging datasets).

Either way, when working with data we are likely to do one of the following:

- Modify existing observations in a data frame
- Create new variables and/or deleting existing ones
- Add new observations and/or deleting existing ones

As always, we will proceed through examples.

3.1 Update values within a data frame

Updating individual observations in a data frame is done using a syntax similar to that of updating values in a list or dictionary.

All we need is the index or key of the value(s) we want to modify and the new desired value(s).

Using our players data frame, let's say we want to change the name and email of Nadal to "Rafa" and "Rafa.Nadal@tennis.com".

[26]:
```
# Update Nadal's name and email
df_players.loc[df_players["surname"] == "Nadal", ["name", "email"]] = ["Rafa",␣
↪"Rafa.Nadal@tennis.com"]
df_players
```

[26]:

	name	surname	email
0	Luca	Rondina	L.Rondina@sussex.ac.uk
1	Roger	Federer	R.Federer@tennis.com
2	Rafa	Nadal	Rafa.Nadal@tennis.com
3	Novak	Djokovic	N.Djokovic@tennis.com

The assignment above is doing three things:

1. Getting the index location via the condition `df_players["surname"] == "Nadal"`
2. Creating a smaller data frame with name and email only using `loc[Boolean_index, ["name", "email"]]`
3. Substituting the values via the assignment operator `=`

3.1.1 Update multiple values

Similarly to dictionaries, data frames have an `update()` method that allows to substitute existing values with new ones whenever the keys match. In a data frame context, the keys will be the variable names.

Let's say that we want to change the emails of all players to the `Name.Surname\@domain` format. We can do that by creating a **new** data frame with the desired email addresses and use the `update()` method on the original data frame.

[27]:
```
# Create a new data frame with email addresses only
df_new_emails = pd.DataFrame({"email": ["Luca.Rondina@surrey.ac.uk", "Roger.
↪Federer@tennis.com", "Rafael.Nadal@tennis.com", "Novak.Djokovic@tennis.
↪com"]})
df_new_emails
```

[27]:

	email
0	Luca.Rondina@surrey.ac.uk
1	Roger.Federer@tennis.com
2	Rafael.Nadal@tennis.com
3	Novak.Djokovic@tennis.com

[28]:
```
# Update the old data frame with new email addresses
df_players.update(df_new_emails)
df_players
```

[28]:

	name	surname	email
0	Luca	Rondina	Luca.Rondina@surrey.ac.uk
1	Roger	Federer	Roger.Federer@tennis.com
2	Rafa	Nadal	Rafael.Nadal@tennis.com
3	Novak	Djokovic	Novak.Djokovic@tennis.com

3.1.2 Modify values directly: the `apply()` method

Sometimes there are cases in which you want to modify the values directly by using some kind of transformation.

Data frames have a method called `apply()` that allows us to apply any transformation stored in a function to the values in the data frame.

As an example, let's say that we want to convert all email addresses to a lowercase version.

[29]:
```
# First define a function that converts strings to lowercase
def lowercase_email(email):
    return email.lower()

# Then apply the function to all email addresses in the data frame
df_players['email'].apply(lowercase_email)
```

[29]:
```
0    luca.rondina@surrey.ac.uk
1     roger.federer@tennis.com
2      rafael.nadal@tennis.com
3    novak.djokovic@tennis.com
Name: email, dtype: object
```

[30]:
```
# Finally update the original data frame
df_players['email'] = df_players['email'].apply(lowercase_email)
df_players
```

[30]:

	name	surname	email
0	Luca	Rondina	luca.rondina@surrey.ac.uk
1	Roger	Federer	roger.federer@tennis.com
2	Rafa	Nadal	rafael.nadal@tennis.com
3	Novak	Djokovic	novak.djokovic@tennis.com

3.2 Adding new variables and observations

Finally, let's see how we can add additional variables and/or observations to our data frame.

3.2.1 Adding new variables

Adding new variables, i.e. columns, is identical to adding a new key and associated values to a dictionary.

For instance, if we want to add the nationality of our players we can simply do that with

[31]:
```python
# Add variable to the data frame
df_players["nationality"] = ["Italian", "Swiss", "Spanish", "Serbian"]
df_players
```

[31]:

	name	surname	email	nationality
0	Luca	Rondina	luca.rondina@surrey.ac.uk	Italian
1	Roger	Federer	roger.federer@tennis.com	Swiss
2	Rafa	Nadal	rafael.nadal@tennis.com	Spanish
3	Novak	Djokovic	novak.djokovic@tennis.com	Serbian

Adding new observations

If instead we want to add new observations to our dataset we can use the data frame `append()` method.

The `append()` method simply enlarges the existing data frame with another new data frame that has the same variable names.

Let's another another couple of legends to our data frame: Pete Sampras and Andre Agassi.

[32]:
```python
# Create a new data frame with new players
df_new_players = pd.DataFrame({
    "name": ["Pete", "Andre"],
    "surname": ["Sampras", "Agassi"],
    "email": ["pete.sampras@tennis.com", "andre.agassi@tennis.com"],
    "nationality": ["American", "American"]
})

# Enlarge existing data frame with the new one
df_players = df_players.append(df_new_players, ignore_index=True)
df_players
```

[32]:

	name	surname	email	nationality
0	Luca	Rondina	luca.rondina@surrey.ac.uk	Italian
1	Roger	Federer	roger.federer@tennis.com	Swiss
2	Rafa	Nadal	rafael.nadal@tennis.com	Spanish
3	Novak	Djokovic	novak.djokovic@tennis.com	Serbian
4	Pete	Sampras	pete.sampras@tennis.com	American
5	Andre	Agassi	andre.agassi@tennis.com	American